Syntactic Analysis.
Write a parser for MiniLan (II)

Theory of Automata and Discrete Mathematics
School of Computer Science
University of Oviedo

1 Exercise 1: New arithmetic operators

The aim of this exercise is to show how the grammar should be changed. Then it will be the time for explaining these changes on the specification files. Finally, after updating the test file, the validation of the parser takes place.

1.1 Introducing subtraction (operator −)

The point is extending the grammar for allowing the subtractions as stated in the language syntactic specification and accomplishing with the precedence among the arithmetic operators.

STEP 1: Introduce the grammar modifications when needed. The MINUS operator (−) for subtracting has the same precedence than the addition operator (+). So, the rule referring to this operation must be included in the same variable: the *arithExpr* variable.

Current grammar	New grammar

`arithExpr -> arithExpr PLUS term`
`  | term ;`
	`arithExpr -> arithExpr PLUS term`
	`  | arithExpr MINUS term`
	`  | term ;`

STEP 2: Modify both the lexical and the syntactic specification files rationally. The **lexical specification file** must be modified for including the new token to generate: the **MINUS** token. So this is the new lexical rule to be included:

```
-   {System.out.println("SCANNER:: MINUS");
     return new Symbol(sym.MINUS);}
```

The **syntactic specification file** requires more changes: a) defining a new *terminal* for the new token -the MINUS terminal-, and b) modifying the *arithExpr* variable's production rule:

```
...
terminal PLUS, MINUS, MULT;
...

arithExpr ::= arithExpr PLUS term  {: System.out.println("PARSER:: arithExp <== arithExpr PLUS term"); :}
            | arithExpr MINUS term {: System.out.println("PARSER:: arithExp <== arithExpr MINUS term"); :}
            | term                 {: System.out.println("PARSER:: arithExp <== term"); :}
            ;
```

STEP 3: Generate the parser. Making use of the script file: `generateParser.bat`

STEP 4: Whenever any error arise, revise the previous steps.

STEP 5: Update the *test* file. To test the parser, the testing program written in MINILAN should be extended with the new language elements that have just been introduced. A test file for the MINUS operator could be:

```
begin
print( 6 - 3);
end
```

STEP 6: Validate the parser: `runParser.bat`. Run the parser and analyze the printed messages: they must correspond with the expected sequence from both the Scanner and the passer

1.2 Introducing the division operator /

With similar reasoning and procedure, extend the arithmetic operations with the division (/). The token **must be DIV**. The printed output to the screen from both the scanner and the parser must be like those for the already implemented arithmetic operators. The test file, this time, might be:

```
begin
print( 6 - 3 / 2.0 );
end
```

1.3 Including parentheses for grouping arithmetic expressions

Currently, the *factor* variable derives a terminal node only (`NUMBER`). However, introducing the parentheses, any arithmetic expression might be a candidate to appear where only `NUMBER` were used to.

The grammar must be modified to allow arithmetic expressions surrounded by parentheses as valid operands, that is, as valid leaves.

Finally, the test file needs updating and the *parser must be validated* with the revised version of the test file. A test file could be as follows:

```
begin
print( (6 - 3) / 2.0 );
end
```

1.4 Allow signed numbers

As stated in the MiniLan syntactic specification file, both the numbers and the expressions in parentheses might be preceded by a sign operator (a + or − symbol). Introduce all the grammar modifications in order to allow these sign operators. A test file could be as follows:

```
begin
print( -(6 - 3) / 2.0 );
end
```